CAPITULO IV

Desarrollo de la Librería PHP

Una vez diseñada la biblioteca, al analizar las partes requeridas para su elaboración, se observa la necesidad de construir un intérprete puro que se encarga de analizar y ejecutar sentencia a sentencia[21] y el mismo va a estar compuesto por un analizador léxico que constituye la primera fase, el cual lee el programa fuente de izquierda a derecha y se agrupa en componentes léxicos (testigos), que son secuencias de caracteres que tienen un significado[22] , un analizador sintáctico que verifica en forma permanente la correcta escritura de las sentencias, teniendo como parámetro un conjunto de reglas denominada "gramática", y un analizador semántico que se encarga de clasificar las palabras según su significado [23] además de un conjunto de clases y funciones que tomaran las sentencias ya evaluadas por el intérprete para obtener como resultado la ejecución de las sentencias a través de una librería[24].

En el desarrollo de la biblioteca, es necesario usar herramientas que sean de ayuda para la elaboración de páginas web, por tal motivo fue seleccionado el lenguaje de programación Php, la misma se lleva a cabo mediante una serie de ciclos o pasos, que están definidos por pruebas y resultados obtenidos en el desarrollo de la metodología Test Driven Development (TDD).

Ciclo de desarrollo de TDD

Elegir Requisito.

Posterior a lo analizado para la creación de la biblioteca, se ha establecido una serie de módulos para la construcción de la misma los cuales se encuentran definidos en la figura 1:

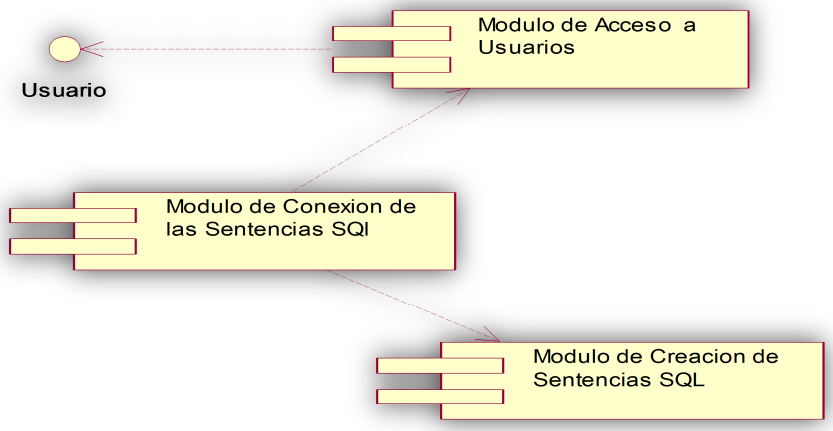


Figura 1 Diagrama de componentes de la biblioteca. Elaboración Propia

Una vez definidos los requisitos generales: módulo de acceso a usuarios, módulo de conexión de las sentencias sql y módulo de creación de creación de sentencias sql, se debe elegir un requisito específico, que es en este caso es Módulo de Creación de Sentencias SQL, donde se desarrolla un intérprete en el que es indispensable construir un analizador léxico, el analizador sintáctico y un analizador semántico, como se muestra en la figura 2:

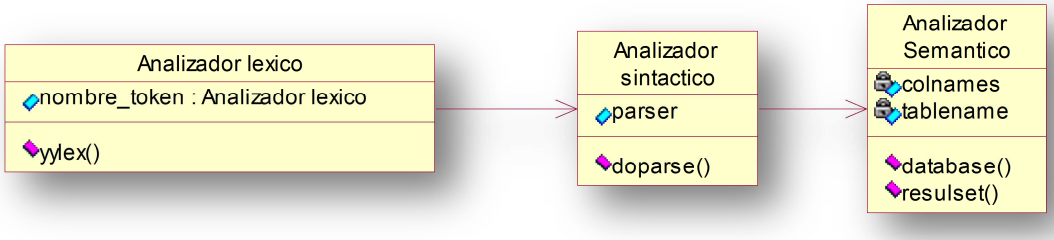


Figura 2. Diagrama de Clases. Elaboración Propia

Analizador Léxico.

El analizador léxico se encarga de leer un flujo de caracteres de entrada y transformarlo en una secuencia de componentes léxicos llamados testigos que son utilizados luego por el analizador sintáctico. Se ocupa de ciertas labores de limpieza entre

ellas: eliminar los espacios en blanco o comentarios, también se ocupa de los problemas que puedan surgir por los distintos juegos de caracteres o si el lenguaje distingue o no las mayúsculas y minúsculas. Se hace uso de JLEXPHP, ya que genera una clase php a partir del archivo léxico construido con JLEX.

Analizador Sintáctico

El analizador sintáctico se encargara de tomar los testigos enviados por el analizador léxico y comprobará que los testigos puedan formar alguna sentencia valida del lenguaje sql.

Analizador Semántico

El analizador semántico se encarga de comprobar que las bases de datos, tablas y campos existan en la misma.

Módulo de Creación de Sentencias

Escribir una Prueba.

Se incluirán dentro del analizador léxico todas las palabras requeridas para la construcción de las sentencias sql. A continuación se muestran las sentencias que son evaluadas por el analizador léxico para el funcionamiento de la biblioteca:

- Select * from tabla
- Select CONCAT(nombre_campo) from tabla where campo1=dato1
- Select UPPER(nombre_campo) from tabla where campo1=dato1
- Select LOWER(nombre_campo) from tabla where campo1=dato1
- Select SUBSTR(nombre_campo) from tabla where campo1=dato1
- Select * from tabla where campo1= dato1 AND campo2=dato2
- Insert into table (campo1, campo2) values (dato1, dato2)
- Update tabla set campo1=dato1, campo2=dato2 where campo3=dato3
- Delete from tabla where campo1=dato1
- Delete from tabla where campo1=dato1 AND campo2=dato2

Una vez identificado el requisito y realizado una breve explicación, se procede a crear el archivo del analizador léxico, donde se introducen los testigos o cadenas entre los cuales están: CREATE, SELECT, FROM, INSERT, UPDATE, DELETE, AND entre otros, que son necesarios en la construcción de las sentencias SQL para el posterior uso de la biblioteca, a continuación se muestran los pasos necesarios para la elaboración del archivo del analizador léxico, comenzando con el archivo vacío hasta completar las especificaciones léxicas necesarias para crear el mismo.

- Inicialmente se procede a crear el archivo you.lex, en este archivo serán introducidos todos los testigos o cadenas que se necesiten para la construcción de la biblioteca.

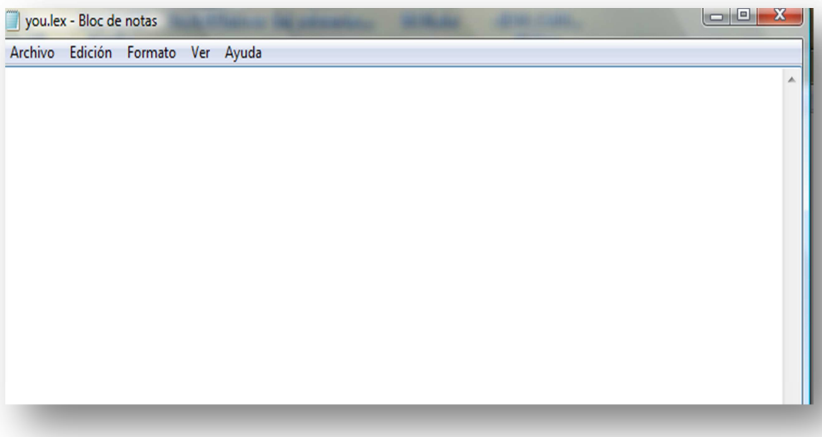


Figura 3 Archivo léxico vacío. Fuente: Elaboración Propia

El resultado obtenido en la creación del archivo es fallido, ya que al ejecutarse se obtendrá como consecuencia una salida vacía, por lo que no se han introducido la serie de caracteres necesarios para el funcionamiento del analizador. Ver figura 4

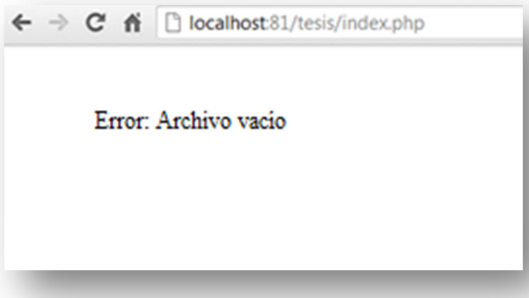


Figura 4 Salida fallida del archivo léxico vacio. Fuente: Elaboración Propia

- Luego en el archivo se realiza un include de jlex.php y la clase PHPLexer, que son requeridas para evaluar la funcionalidad del archivo, como se muestra en la figura 5:

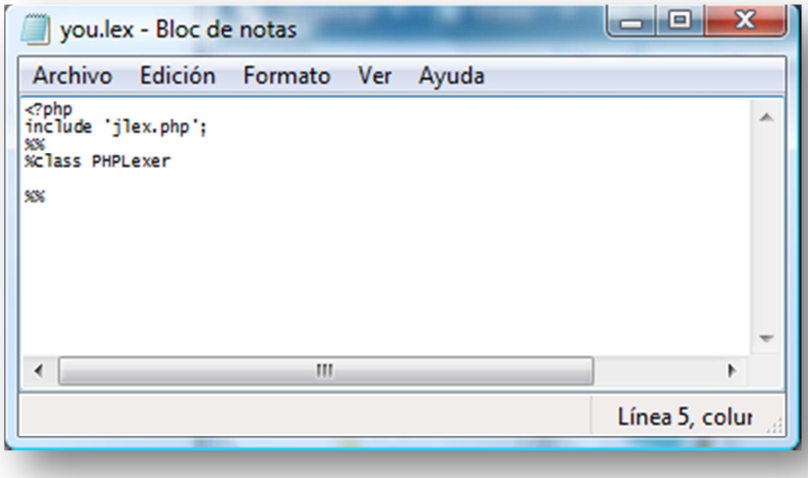


Figura 5 Archivo léxico inicial. Fuente: Elaboración Propia

- En el siguiente paso se introducen los números, caracteres, espacios y saltos de líneas dentro del doble porcentaje que es donde se incluyen las directivas y macros que son empleados para la evaluación de las sentencias SQL.

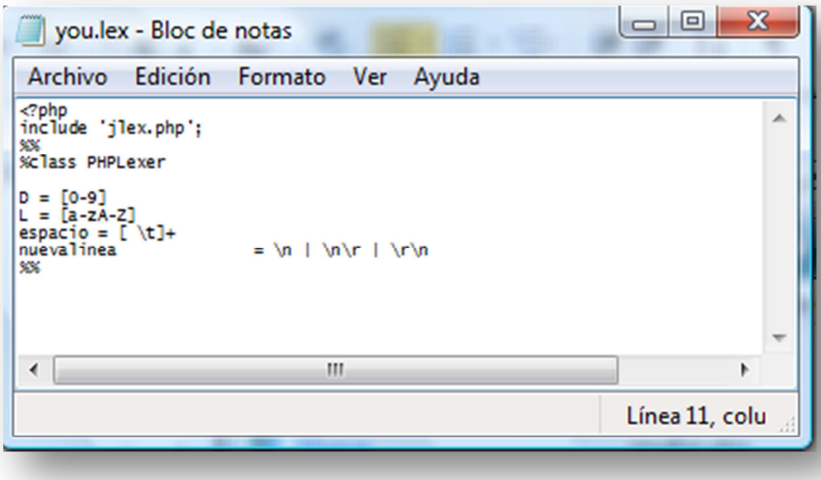


Figura 6 Inclusión de directivas al archivo léxico. Fuente: Elaboración Propia

Debido a que no se han implementado las reglas léxicas en la resolución del archivo, el resultado adquirido hasta este momento es erróneo.

Para brindar una mejor información de los testigos requeridos para la creación del analizador léxico, serán definidos a continuación: Select, From, Where, Or, And, In, Insert, Into, Values, Update, Set, Delete, Create, Drop, Database, Tabla, Integer, Float, Character, Null, Auto_Incremente, (,), “,”, +,*,:,=, espacio, entre otros.

Asimismo, después de haberse identificado los testigos requeridos para la evaluación de las sentencias SQL para este proyecto, se van incorporando de manera progresiva las reglas léxicas tal como se muestra en la figura 7:

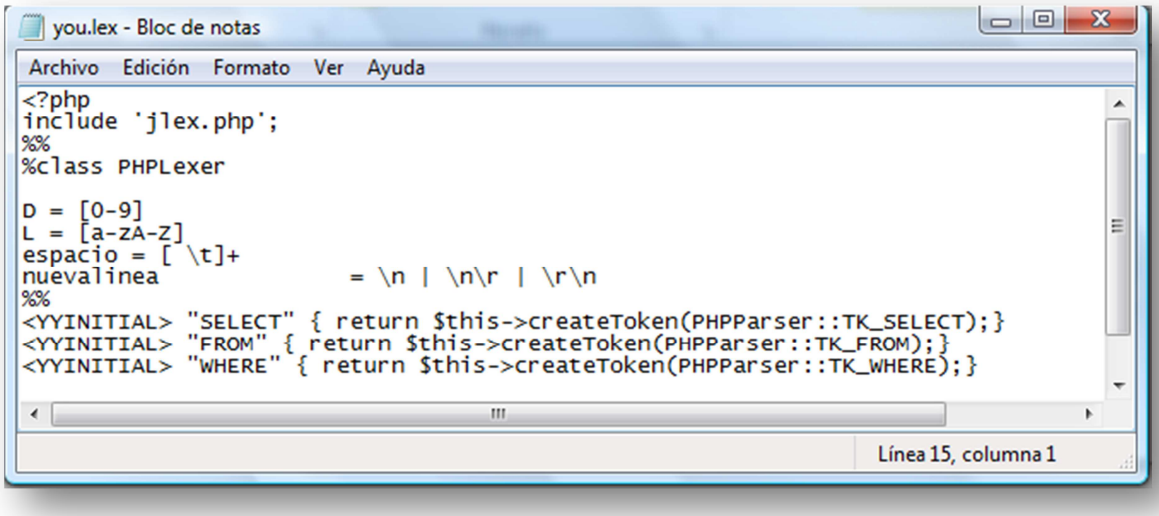


Figura 7 Inclusión de reglas léxicas al archivo. Fuente: Elaboración Propia

En este segmento del archivo se introducen las palabras básicas para la conformación de la sentencia Select, en el que aún no se obtiene un resultado satisfactorio ya que falta la inclusión de la regla léxica correspondiente a un * y otras requeridas para la funcionalidad de la consulta.

En la figura 8 se muestra la incorporación de todas las palabras necesarias para la evaluación de las sentencias SQL, tales como Select, Insert, Update , Delete, Create, Where, entre otros.

```
<?php
include 'jlex.php';
%%
%class PHPLexer

D = [0-9]
L = [a-zA-Z]
espacio = [ \t]+
nuevalinea = \n | \n\r | \r\n
%%
<YYINITIAL> "SELECT" { return $this->createToken(PHPParser::TK_SELECT);}
<YYINITIAL> "FROM" { return $this->createToken(PHPParser::TK_FROM);}
<YYINITIAL> "WHERE" { return $this->createToken(PHPParser::TK_WHERE);}
<YYINITIAL> "<>"|"<"|">"|"<="|">=" { return $this->createToken(PHPParser::TK_COMPARISON);}
<YYINITIAL> "OR" { return $this->createToken(PHPParser::TK_OR);}
<YYINITIAL> "AND" { return $this->createToken(PHPParser::TK_AND);}
<YYINITIAL> "INSERT" { return $this->createToken(PHPParser::TK_INSERT);}
<YYINITIAL> "INTO" { return $this->createToken(PHPParser::TK_INTO);}
<YYINITIAL> "VALUES" { return $this->createToken(PHPParser::TK_VALUES);}
<YYINITIAL> "UPDATE" { return $this->createToken(PHPParser::TK_UPDATE);}
```

```
<YYINITIAL> "SET" { return $this->createToken(PHPParser::TK_SET);}
<YYINITIAL> "DELETE" { return $this->createToken(PHPParser::TK_DELETE);}
<YYINITIAL> "CREATE" { return $this->createToken(PHPParser::TK_CREATE);}
<YYINITIAL> "DROP" { return $this->createToken(PHPParser::TK_DROP);}
<YYINITIAL> "DATABASE" { return $this->createToken(PHPParser::TK_DATABASE);}
<YYINITIAL> "TABLE" { return $this->createToken(PHPParser::TK_TABLE);}
<YYINITIAL> "DEFAULT" { return $this->createToken(PHPParser::TK_DEFAULT);}
<YYINITIAL> "INTEGER" { return $this->createToken(PHPParser::TK_INTEGER);}
<YYINITIAL> "CHARACTER" { return $this->createToken(PHPParser::TK_CHARACTER);}
<YYINITIAL> "NULL" { return $this->createToken(PHPParser::TK_NULL);}
<YYINITIAL> "AUTO_INCREMENT" { return $this->createToken(PHPParser::TK_AUTO_INCREMENT);}
<YYINITIAL> "AS" { return $this->createToken(PHPParser::TK_AS);}
```

Figura 8 Inclusión de testigos al archivo léxico. Elaboración propia

- Una vez ingresadas todos las frases requeridas para realizar las consultas, se integran los operadores comparativos que son utilizados por las sentencias SQL para el buen funcionamiento de la misma.

```
<?php
include 'jlex.php';
%%
%class PHPLexer

D = [0-9]
L = [a-zA-Z]
espacio = [ \t]+
nuevalinea                      = \n | \n\r | \r\n
%%
<YYINITIAL> "SELECT" { return $this->createToken(PHPParser::TK_SELECT);}
<YYINITIAL> "FROM" { return $this->createToken(PHPParser::TK_FROM);}
<YYINITIAL> "WHERE" { return $this->createToken(PHPParser::TK_WHERE);}
<YYINITIAL> "<>"|"<"|">"|"<="|">=" { return $this->createToken(PHPParser::TK_COMPARISON);}
<YYINITIAL> "OR" { return $this->createToken(PHPParser::TK_OR);}
<YYINITIAL> "AND" { return $this->createToken(PHPParser::TK_AND);}
<YYINITIAL> "INSERT" { return $this->createToken(PHPParser::TK_INSERT);}
<YYINITIAL> "INTO" { return $this->createToken(PHPParser::TK_INTO);}
<YYINITIAL> "VALUES" { return $this->createToken(PHPParser::TK_VALUES);}
<YYINITIAL> "UPDATE" { return $this->createToken(PHPParser::TK_UPDATE);}
<YYINITIAL> "SET" { return $this->createToken(PHPParser::TK_SET);}
<YYINITIAL> "DELETE" { return $this->createToken(PHPParser::TK_DELETE);}
<YYINITIAL> "CREATE" { return $this->createToken(PHPParser::TK_CREATE);}
<YYINITIAL> "DROP" { return $this->createToken(PHPParser::TK_DROP);}
<YYINITIAL> "DATABASE" { return $this->createToken(PHPParser::TK_DATABASE);}
<YYINITIAL> "TABLE" { return $this->createToken(PHPParser::TK_TABLE);}
<YYINITIAL> "DEFAULT" { return $this->createToken(PHPParser::TK_DEFAULT);}
<YYINITIAL> "INTEGER" { return $this->createToken(PHPParser::TK_INTEGER);}
<YYINITIAL> "CHARACTER" { return $this->createToken(PHPParser::TK_CHARACTER);}
<YYINITIAL> "NULL" { return $this->createToken(PHPParser::TK_NULL);}
<YYINITIAL> "AUTO_INCREMENT" { return $this->createToken(PHPParser::TK_AUTO_INCREMENT);}
<YYINITIAL> "AS" { return $this->createToken(PHPParser::TK_AS);}
<YYINITIAL> {D}*\.?({D})+ { return $this->createToken(PHPParser::TK_FLOATNUM);}
<YYINITIAL> ({D})+ { return $this->createToken(PHPParser::TK_INTNUM);}
<YYINITIAL> {L}({L}|{D})* { return $this->createToken(PHPParser::TK_NAME);}
<YYINITIAL> L?\'(\\.|[^\\\'])*\' { return $this->createToken(PHPParser::TK_CHARVAL);}
<YYINITIAL> L?\"(\\.|[^\\\"])*\" { return $this->createToken(PHPParser::TK_STRING);}
<YYINITIAL> "(" { return $this->createToken(PHPParser::TK_LPAREN);}
<YYINITIAL> ")" { return $this->createToken(PHPParser::TK_RPAREN);}
<YYINITIAL> "," { return $this->createToken(PHPParser::TK_COMMA);}
```

```
<YYINITIAL> ":" { return $this->createToken(PHPParser::TK_COLON);}
<YYINITIAL> "." { return $this->createToken(PHPParser::TK_PUNTO);}
<YYINITIAL> ";" { return $this->createToken(PHPParser::TK_SEMIC);}
<YYINITIAL> "+" { return $this->createToken(PHPParser::TK_PLUS);}
<YYINITIAL> "-" { return $this->createToken(PHPParser::TK_MINUS);}
<YYINITIAL> "*" { return $this->createToken(PHPParser::TK_STAR);}
<YYINITIAL> "/" { return $this->createToken(PHPParser::TK_SLASH);}
<YYINITIAL> "=" { return $this->createToken(PHPParser::TK_EQUAL);}
<YYINITIAL> [ \t\v\n\f\r] { }
<YYINITIAL> {nuevalinea}    { }
<YYINITIAL> {espacio}    { }
```

Figura 9 Archivo final del analizador léxico. Elaboración propia

Luego de haber ingresado las reglas léxicas necesarias para el buen funcionamiento del archivo, se obtiene un resultado satisfactorio y para comprobarlo se genera una clase que va hacer usada durante la ejecución de las sentencias, dicha clase estará implementada en Php, la cual fue creada con el nombre de UnetLexer.php, a través del comando Java –cp JlexPhp.Main you.lex, que se ejecuta por medio de la consola al no generar ningún error procede a generar el archivo que se muestra en la figura 10:

```
<?php
include 'jlex.php';


class PHPLexer extends JLexBase  {
  const YY_BUFFER_SIZE = 512;
  const YY_F = -1;
  const YY_NO_STATE = -1;
  const YY_NOT_ACCEPT = 0;
  const YY_START = 1;
  const YY_END = 2;
  const YY_NO_ANCHOR = 4;
  const YY_BOL = 128;
  var $YY_EOF = 129;

  function    construct($stream) {
            parent::__construct($stream);
            $this->yy_lexical_state = self::YYINITIAL;
  }

  const YYINITIAL = 0;
  static $yy_state_dtrans = array(
            0
  );
  static $yy_acpt = array(
            /* 0 */ self::YY_NOT_ACCEPT,
            /* 1 */ self::YY_NO_ANCHOR,
            /* 2 */ self::YY_NO_ANCHOR,
            /* 3 */ self::YY_NO_ANCHOR,
                        •
```

•
•
•

Figura 10 Archivo generado del analizador léxico. Elaboración propia

Luego de elaborar el archivo y colocar en funcionamiento la prueba de creación del lenguaje léxico, se debe hacer uso del siguiente código php, ya que este tiene la labor de desempeñarse como una función principal:

```
<?
include 'UnetLexer.php';
$scanner =new PHPlexer (fopen("prueba.txt","r"));
while($t=$scanner->yylex()){
  echo "<br/>";
  print_r($t); echo "Nombre del Token -> ".$scanner->yy_token_name($t->type);
}
?>
```

Después se procede a crear el código correspondiente que ejecutará los diferentes testigos, mediante un archivo de texto en el que se guardan las palabras que componen las sentencias sql propuestas para este proyecto.

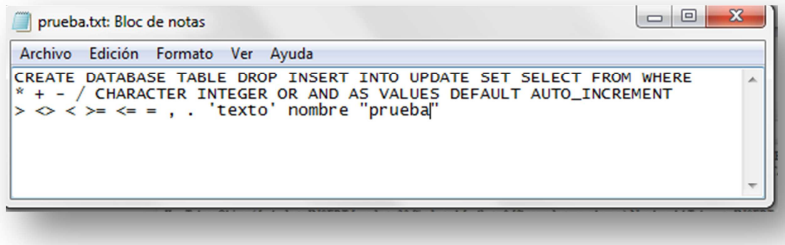

prueba.txt: Bloc de notas
Archivo Edición Formato Ver Ayuda
CREATE DATABASE TABLE DROP INSERT INTO UPDATE SET SELECT FROM WHERE
* + - / CHARACTER INTEGER OR AND AS VALUES DEFAULT AUTO_INCREMENT
> <> < >= <= = , . 'texto' nombre "prueba"

Figura 11. Archivo de prueba del analizador léxico. Fuente: Elaboración propia

Los resultados logrados con la creación del analizador léxico se dieron de manera

favorable debido a que en el mismo se ilustran los testigos requeridos para la realización de las sentencias sql que serán soportadas por la biblioteca como se puede observar en la figura 12:

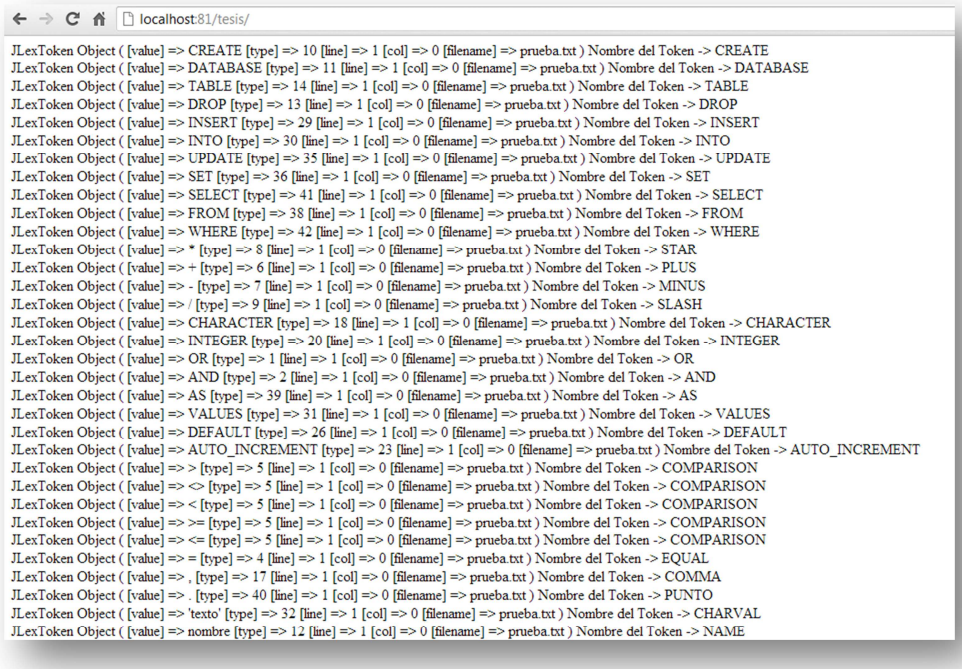


Figura 12. Resultado de los testigos del léxico. Fuente: Elaboración propia

Una vez realizado el requisito de la construcción del analizador léxico, el siguiente paso es la elaboración del analizador sintáctico, ya que este toma los testigos que le envía el analizador léxico y comprueba que los testigos pueden formar alguna sentencia valida del lenguaje. Para evaluar la sintaxis del lenguaje de programación habitualmente se describe por gramáticas libres de contexto, utilizando para ello un tipo de notación como por ejemplo notación BNF donde se representan reglas de escritura o producciones. La gramática o sintaxis determinan las cadenas de símbolos correctos o sentencias para formar el lenguaje.

De esta manera la creación del analizador sintáctico es importante ya que este se

encarga de evaluar y verificar que las sentencias sql en la elaboración de la biblioteca se encuentre estructurada de forma correcta, y para ello se realizará una serie de pruebas donde se comprobará el funcionamiento de cada una de las sentencias propuestas en el desarrollo de esta investigación.

Primeramente se elabora el archivo you.y, donde se van a introducir las reglas de producción para la evaluación de la sintaxis correspondientes a cada sentencia sql, comenzando con el archivo vacío hasta completar todas las reglas de producción que comprobaran que las sentencias se encuentren escritas correctamente.

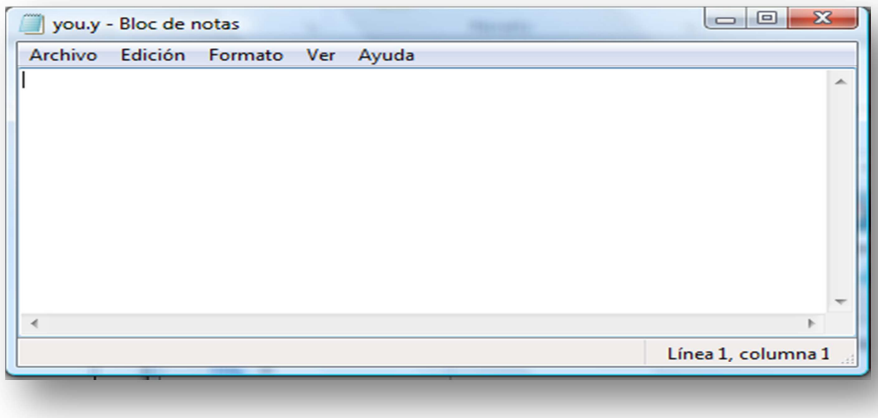


Figura 13 Archivo del analizador Sintáctico vacío. Fuente: Elaboración Propia

Es importante destacar que como aún no se han introducido las reglas de producción necesarios para la evaluación de la sintaxis, el resultado obtenido durante la ejecución de las sentencias anteriormente expuestas es fallido. Ver figura 14

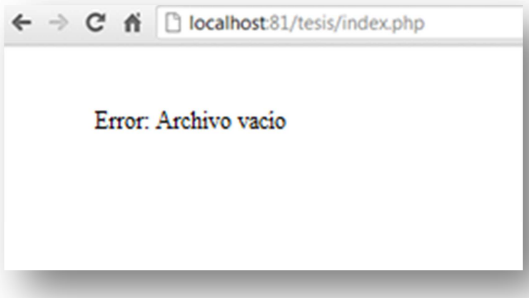


Figura 14 Salida fallida del archivo sintáctico. Fuente: Elaboración Propia

En la figura 15 se muestra la inclusión de las librerías o archivos usados en la evaluación de la sintaxis, se comprueba si se produce algún error al momento de ejecutar la sentencia, se crea una pequeña función que muestra que el análisis se haya realizado completo, y finalmente la precedencia de algunos símbolos usados para la ejecución de las sentencias sql.

Esta estructura del archivo junto con las reglas de producción se usaran para la ejecución de todas las sentencias sql empezando por las consultas de selección (select), Insert, Update y finalizando con la consulta de eliminación (delete), que serán evaluadas de forma unitaria. Ver a continuación la estructura del archivo:

```
%name PHPParser
%token_prefix TK_
%token_type {int}
%include {
}
  %syntax_error {
 if($yymajor==0)$message="Unexpected End";
 throw new Exception($message);
 return false;
}
%parse_accept {
        echo "<br/><div style='padding: 8px; margin-top: 1em; background-color: green; color: white;'>
<strong>Analisis Completo!</strong></div>";
    return true;
}
%left OR.
%left AND.
%left NOT.
%left EQUAL COMPARISON.
%left PLUS MINUS.
%left STAR SLASH.
```

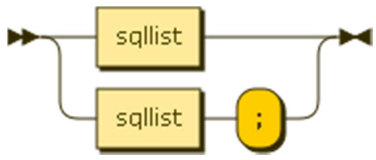
Figura 15 Inclusión de archivos con precedencias. Elaboración Propia

Una vez creada la parte inicial del archivo, se proceden a elaborar las reglas de producción para la ejecución de las diferentes sentencias, en el que se evaluará la estructura que compone la regla. Para ofrecer un mejor entendimiento de la creación de las reglas se ilustraran los distintos diagramas de conway correspondientes a cada consulta.

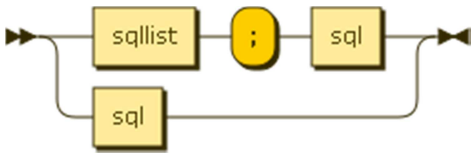
A continuación se muestran las sentencias de ejecución de las principales consultas de Sql para el lenguaje de manipulación de datos (dml):

- **Select:**

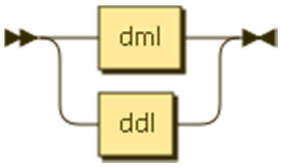
input:



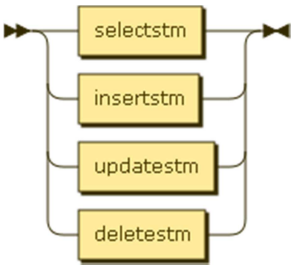
sqllist:



sql:



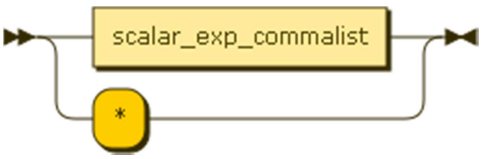
dml:



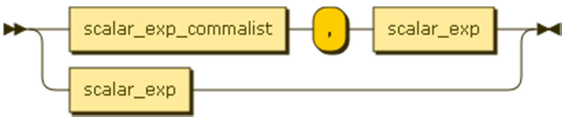
selectstm:

query_spec:

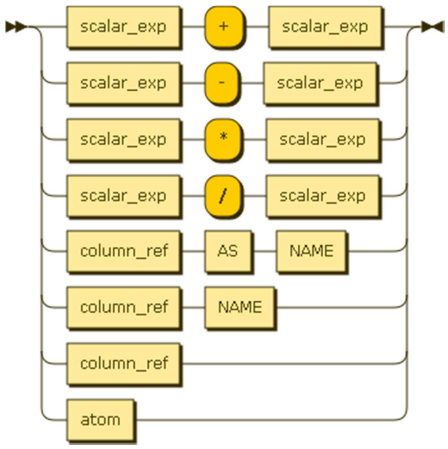
selection:



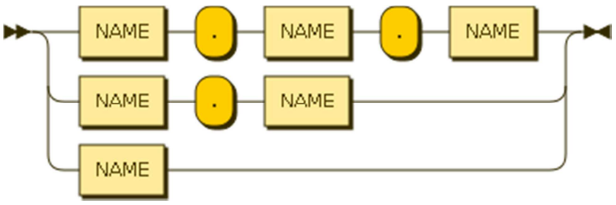
scalar_exp_commalist:



scalar_exp:

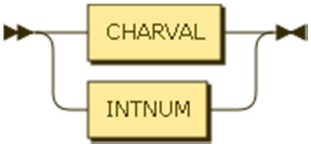


column_ref:

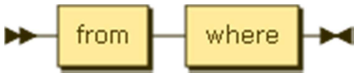


atom:

literal:

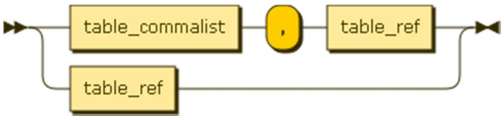


table_exp:

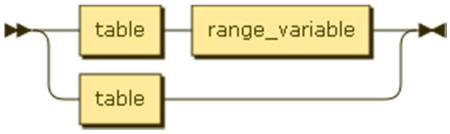


from:

table_commalist:

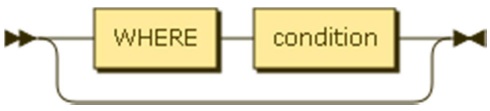


table_ref:



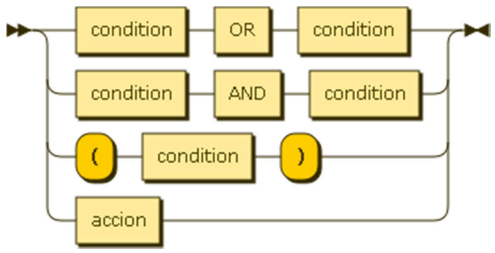
range_variable:

where:



condition:

accion:



comparacion:

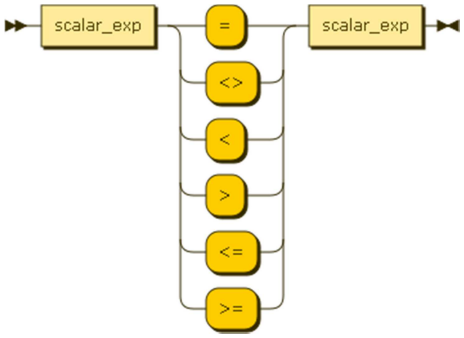


Tabla 1 Diagrama de Conway para sentencia SELECT. Elaboración Propia

El archivo proporcionado es:

```
input::= sqllist.
input::= sqllist SEMIC.
sqllist ::= sqllist SEMIC sql.
sqllist ::= sql.
sql ::= dml.
dml ::= selectstm.
selectstm ::= query_spec.
query_spec ::= SELECT selection table_exp.
selection ::= scalar_exp_commalist.
selection ::= STAR.
scalar_exp_commalist ::= scalar_exp_commalist COMMA scalar_exp.
scalar_exp_commalist ::= scalar_exp.
scalar_exp ::= scalar_exp PLUS scalar_exp.
scalar_exp ::= scalar_exp MINUS scalar_exp.
scalar_exp ::= scalar_exp STAR scalar_exp.
scalar_exp ::= scalar_exp SLASH scalar_exp.
scalar_exp ::= column_ref AS NAME.
scalar_exp ::= column_ref NAME.
scalar_exp ::= column_ref.
scalar_exp ::= atom.
column_ref ::= NAME PUNTO NAME PUNTO NAME.
column_ref ::= NAME PUNTO NAME.
column_ref ::= NAME.
atom ::= literal.
literal ::= CHARVAL.
literal ::= INTNUM.
table_exp ::= from where.
from ::= FROM table_commalist.
```

```
table  commalist  ::= table  commalist COMMA table_ref.
table_commalist  ::= table_ref.
table_ref ::= table range_variable.
table_ref ::= table.
table ::= NAME PUNTO NAME.
table ::= NAME.
range_variable ::= NAME.
where ::= WHERE condition.
where ::= .
condition ::= condition OR condition.
condition ::= condition AND condition.
condition ::= LPAREN condition RPAREN.
condition ::= accion.
accion ::= comparacion.
comparacion ::= scalar_exp EQUAL scalar_exp.
comparacion ::= scalar_exp COMPARISON scalar_exp.
atom_commalist ::= atom_commalist COMMA atom.
atom_commalist ::= atom.
```

Figura 16 Reglas de producción de la sentencia Select. Elaboración Propia

Para obtener la ejecución de los distintos casos de prueba, se debe hacer uso del siguiente código elaborado en el lenguaje php, donde se incluye el archivo generado por el analizador léxico llamado UnetLexer.php y el archivo creado por el analizador sintáctico.

Para generar el analizador sintáctico, es necesario utilizar el siguiente comando: lemon –LPHP you.y, obteniendo de esta forma el parser, llamándose al archivo UnetParser.php. Este código será aplicado a todas las pruebas para la obtención de los resultados.

```php
<?
include 'UnetParser.php';
include 'UnetLexer.php';
$parser =new PHPParser();
$scanner =new PHPlexer (fopen("prueba.txt","r"));
while($t=$scanner->yylex()){
  $parser->doParse($t->type,$t);
}
$parser->doParse(0);
?>
```

Para evaluar el funcionamiento del archivo creado con las reglas de producción para la sentencia select, se harán uso de los diferentes tipos de selects que se muestran a continuación:

```
SELECT * FROM tabla;
```

```
SELECT * FROM tabla WHERE a=1;
SELECT a,b,c FROM tabla;
SELECT a,b,c FROM tabla WHERE a=1;
SELECT c FROM tabla1,tabla2 WHERE a=1 AND b='2';
SELECT tabla2.c FROM tabla1,tabla2 WHERE a=1 AND b='2';
```

Figura 17 Ejemplos para la sentencia Select. Elaboración Propia

Donde el resultado conseguido con la ejecución de cada uno de ellos es exitoso tal como se muestra en la figura 18:

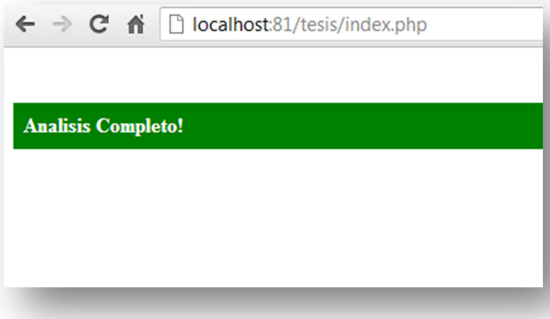


Figura 18 Resultado de la sentencia Select. Fuente: Elaboración Propia

- **Insert:**

La siguiente prueba en elaborarse es la inclusión de las reglas para la evaluación de la sentencia Insert, del lenguaje de manipulación de datos. Al igual que para la sentencia Select, se evalúa la buena estructuración de la sentencia. A continuación se ilustran los diagramas de conway correspondientes:

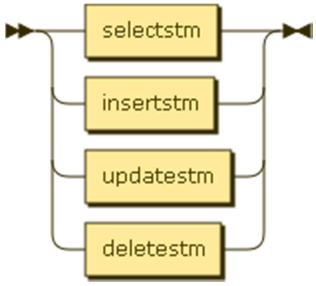
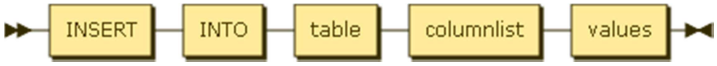
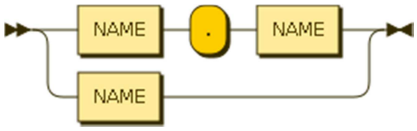
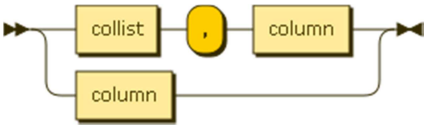
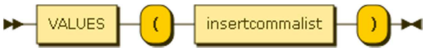
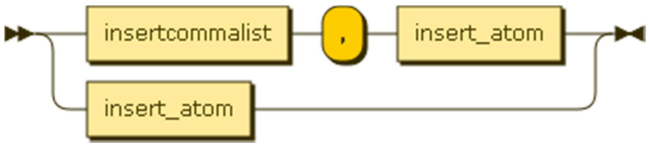
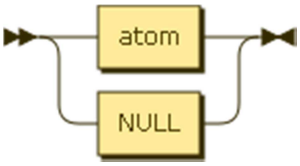
<table>
<tr><td>dml:
 
</td><td>insertstm:
 
</td></tr>
<tr><td>table:
 
</td><td>columlist:
</td></tr>
<tr><td>collist:
 
</td><td>column:
</td></tr>
<tr><td>values:
 
</td><td>insertcommalist:
 
</td></tr>
<tr><td>insert_atom:
 
</td><td></td></tr>
</table>

Tabla 2 Diagrama de Conway para sentencia INSERT. Elaboración Propia

En la figura 19 se muestra la inclusión de las reglas de producción para la sentencia Insert:

```
input::= sqllist.
input::= sqllist SEMIC.
sqllist ::= sqllist SEMIC sql.
sqllist ::= sql.
sql ::= dml.
dml ::= insertstm.
insertstm ::= INSERT INTO table columnlist values.
table ::= NAME PUNTO NAME.
table ::= NAME.
columnlist ::= LPAREN collist RPAREN.
collist ::= collist COMMA column.
collist ::= column.
column ::= NAME.
values ::= VALUES LPAREN insertcommalist RPAREN.
insertcommalist ::= insertcommalist COMMA insert_atom.
insertcommalist ::= insert_atom.
insert_atom ::= atom.
insert_atom ::= NULL.
atom ::= literal.
literal ::= CHARVAL.
literal ::= INTNUM.
```

Figura 19 Reglas de producción de la sentencia Insert. Elaboración Propia

El caso de prueba realizado para este conjunto de reglas es:

```
INSERT INTO tabla (campo1,campo2) VALUES (1,'2')
```

Figura 20 Ejemplo para la sentencia Insert. Elaboración Propia

Y el resultado que se obtuvo como ejecución de la sentencia Sql para el Insert es exitoso tal como se ilustra en la figura 21:

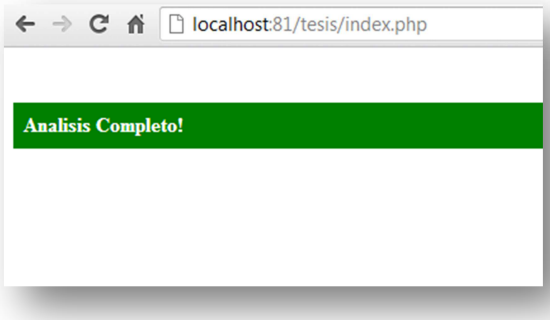


Figura 21 Resultado de la sentencia Insert. Fuente: Elaboración Propia

- **Update:**

Para la creación de la sentencia Update, se elaboraron las reglas que componen la estructura de la misma, haciéndose el procedimiento similar al que se realizó para la evaluación de las sentencias anteriores. Se Muestra a continuación los diagramas de conway:

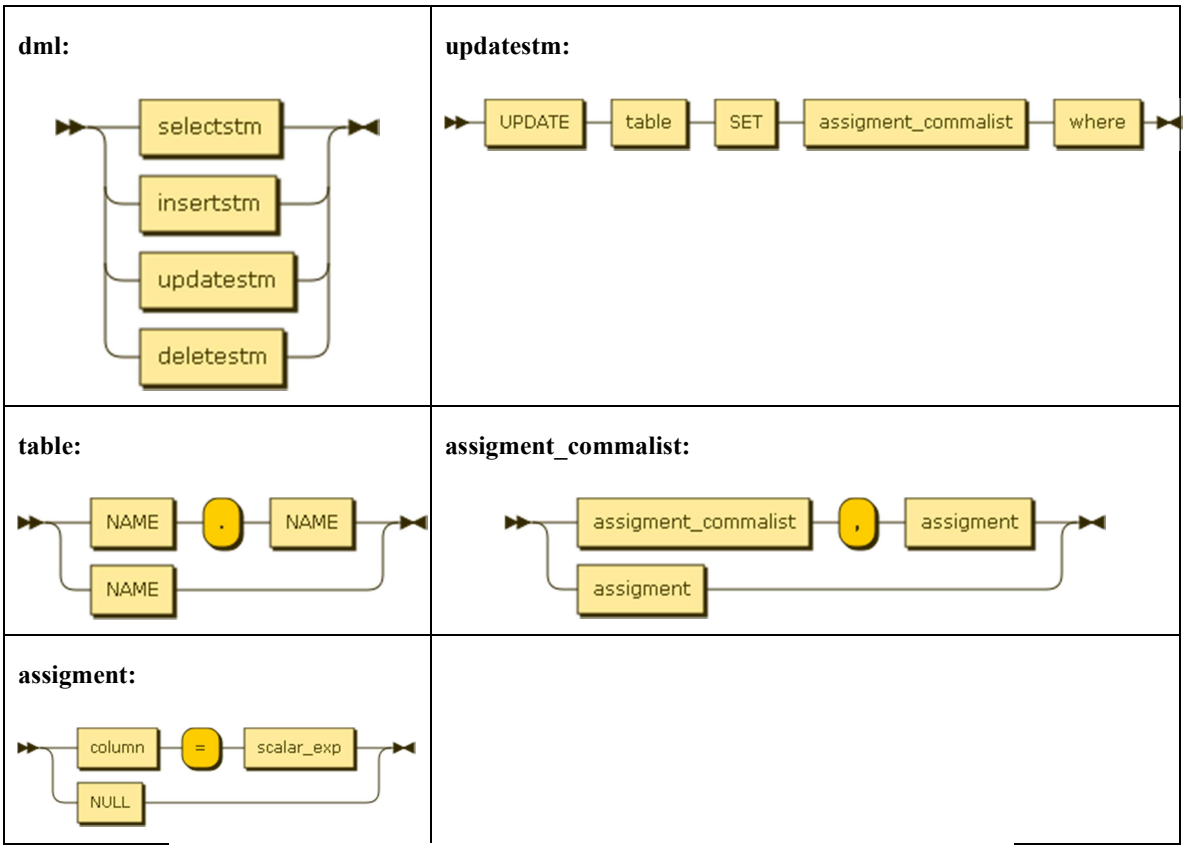


Tabla 3 Diagrama de Conway para sentencia UPDATE. Elaboración Propia

A continuación se ilustra la figura 25:

```
input::= sqllist.
input::= sqllist SEMIC.
sqllist ::= sqllist SEMIC sql.
sqllist ::= sql.
sql ::= dml.
dml ::= updatestm.
updatestm ::= UPDATE table SET assigment_commalist where.
table ::= NAME PUNTO NAME.
table ::= NAME.
assigment_commalist ::= assigment_commalist COMMA assigment.
assigment_commalist ::= assigment.
assigment ::= column EQUAL scalar_exp.
assigment ::= NULL.
column ::= NAME.
scalar_exp ::= scalar_exp PLUS scalar_exp.
scalar_exp ::= scalar_exp MINUS scalar_exp.
scalar_exp ::= scalar_exp STAR scalar_exp.
scalar_exp ::= scalar_exp SLASH scalar_exp.
```

```
scalar_exp ::= column_ref AS NAME.
scalar_exp ::= column_ref NAME.
scalar_exp ::= column_ref.
scalar_exp ::= atom.
column_ref ::= NAME PUNTO NAME PUNTO NAME.
column_ref ::= NAME PUNTO NAME .
column_ref ::= NAME.
atom ::= literal.
literal ::= CHARVAL.
literal ::= INTNUM.
where ::= WHERE condition.
where ::= .
condition ::= condition OR condition.
condition ::= condition AND condition.
condition ::= LPAREN condition RPAREN.
condition ::= accion.
accion ::= comparacion.
comparacion ::= scalar  exp EQUAL scalar  exp.
comparacion ::= scalar_exp COMPARISON scalar_exp.
atom  commalist ::= atom _commalist COMMA atom.
atom_commalist ::= atom.
```

Figura 22 Reglas de producción de la sentencia Update. Elaboración Propia

La prueba que se realizó para la comprobación de las reglas es la siguiente:

```
UPDATE tabla SET campo=1, campo2='2' WHERE campo3=4;

UPDATE tabla SET estatus='1';
```

Figura 23 Ejemplos de la sentencia Update. Elaboración Propia

A continuación se muestra en la figura 24 el resultado exitoso obtenido con la ejecución de las sentencias Update:

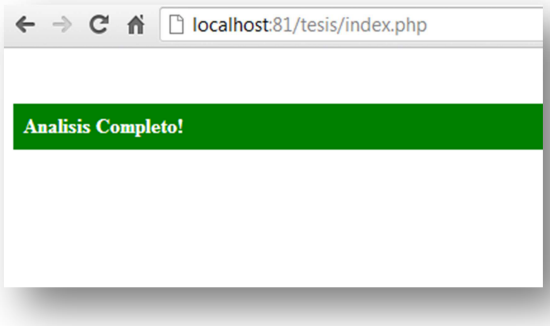


Figura 24 Resultado de la sentencia Update. Fuente: Elaboración Propia

- **Delete:**

En la elaboración de la sentencia Delete, se realizó el mismo procedimiento, se incluyen las reglas de la gramática que evaluaran la estructura para este tipo de sentencia, a continuación se representan los diagramas:

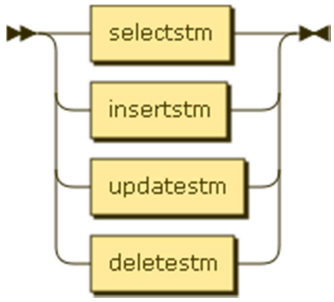
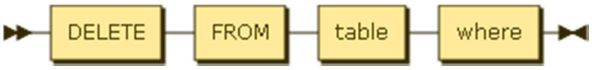
dml:	**deletestm:**
	

Tabla 4 Diagrama de Conway para sentencia UPDATE. Elaboración Propia

La figura 25 corresponde a las reglas de la sentencia Delete:

```
input::= sqllist.
input::= sqllist SEMIC.
sqllist ::= sqllist SEMIC sql.
sqllist ::= sql.
sql ::= dml.
dml ::= deletestm.
deletestm ::= DELETE FROM table where.
table ::= NAME PUNTO NAME.
table ::= NAME.
where ::= WHERE condition.
where ::= .
condition ::= condition OR condition.
condition ::= condition AND condition.
condition ::= LPAREN condition RPAREN.
condition ::= accion.
accion ::= comparacion.
comparacion ::= scalar_exp EQUAL scalar_exp.
comparacion ::= scalar  exp COMPARISON scalar_exp.
scalar_exp ::= scalar_exp PLUS scalar_exp.
scalar_exp ::= scalar_exp MINUS scalar_exp.
scalar_exp ::= scalar_exp STAR scalar_exp.
scalar_exp ::= scalar_exp SLASH scalar_exp.
scalar_exp ::= column_ref AS NAME.
scalar_exp ::= column_ref NAME.
scalar_exp ::= column_ref.
scalar_exp ::= atom.
column_ref ::= NAME PUNTO NAME PUNTO NAME.
column_ref ::= NAME PUNTO NAME .
column_ref ::= NAME.
atom_commalist ::= atom_commalist COMMA atom.
atom_commalist ::= atom.
atom ::= literal.
```

```
literal ::= CHARVAL.
literal ::= INTNUM.
```

Figura 25 Gramática libre de contexto Delete. Elaboración Propia

Para realizar la comprobación de la estructura de la sentencia se aplicó la siguiente prueba:

```
DELETE FROM tabla  WHERE campo3=4;

DELETE FROM tabla  ;
```

Figura 26 Ejemplos de la sentencia Update. Elaboración Propia

El resultado sintácticamente obtenido es correcto, a continuación se ilustra en la figura 27:

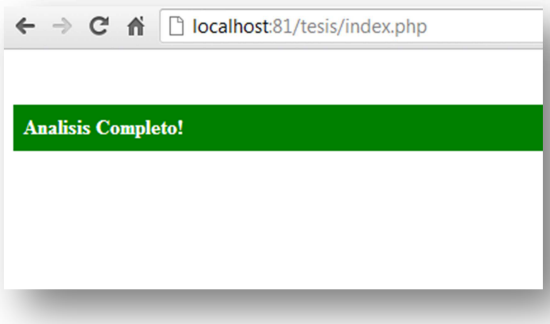
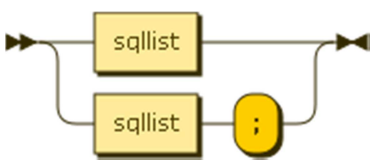
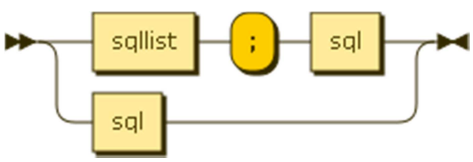


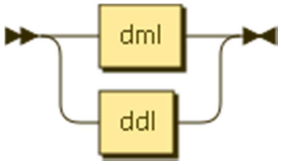
Figura 27 Resultado de la sentencia Delete. Fuente: Elaboración Propia

A continuación se muestra la ejecución de las principales consultas de Sql para el lenguaje de definición de datos (ddl), por medio de diagramas de conway:

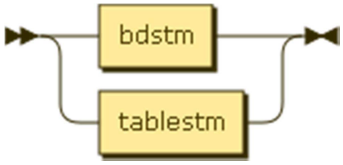
- **Create y Drop**

input:	**sqllist:**
	

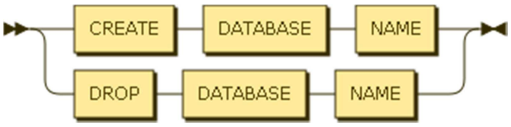
sql:



ddl:



bdstm:



tablestm:

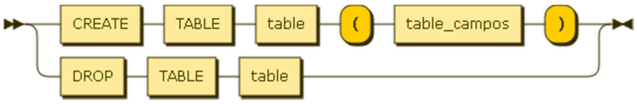
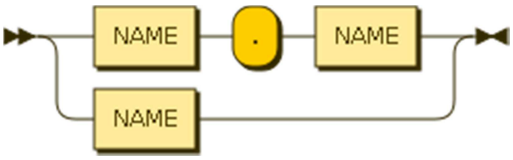
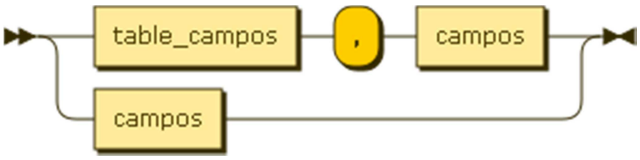


table:



table_campos:

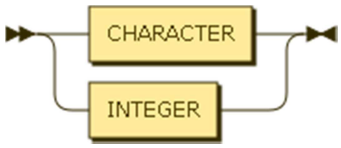


campos:

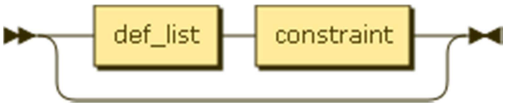
column_def:

column:

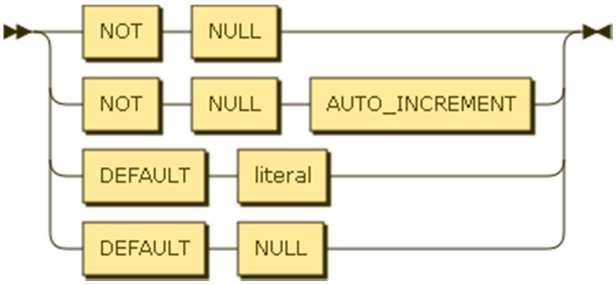
tipo_dato:



def_list:



constraint:



literal:

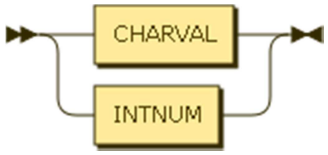


Tabla 5 Diagrama de Conway para sentencias DDL. Elaboración Propia

A continuación se muestra la figura 28:

```
input:= sqllist.
input:= sqllist SEMIC.
sqllist ::= sqllist SEMIC sql.
sqllist ::= sql.
sql ::= dml.
sql ::= ddl.
ddl ::= bdstm.
ddl ::= tablestm.
bdstm ::= CREATE DATABASE NAME.
bdstm ::= DROP DATABASE NAME.
tablestm  ::= CREATE TABLE table LPAREN table_campos RPAREN.
tablestm ::= DROP TABLE table.
table_campos ::=  table_campos COMMA campos.
table_campos ::= campos.
campos ::=  column_def.
column_def ::= column tipo_dato def_list.
tipo  dato ::=  CHARACTER.
tipo_dato ::=  INTEGER.
def_list ::=  def_list constraint.
def_list ::= .
constraint ::= NOT NULL.
constraint ::= NOT NULL AUTO_INCREMENT.
constraint ::= DEFAULT literal.
constraint ::= DEFAULT NULL.
column ::= NAME.
table ::= NAME PUNTO NAME.
table ::= NAME.
literal ::= CHARVAL.
literal ::= INTNUM.
```

Figura 28 Reglas de producción de las sentencias DDL. Elaboración Propia

El caso de prueba realizado para este conjunto de reglas es:

```
CREATE DATABASE prueba

CREATE TABLE table (campo1 INTEGER NOT NULL AUTO_INCREMENT,campo2 CHARACTER DEFAULT '1')

DROP TABLE table

DROP DATABSE prueba
```

Figura 29 Ejemplo para la sentencias CREATE y DROP. Elaboración Propia

Y el resultado obtenido en la ejecución de la sentencia Sql para crear y eliminar es exitoso como se ilustra en la figura 30:

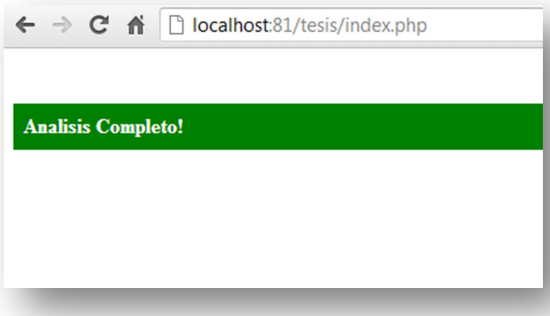


Figura 30 Resultado de la sentencia DDL. Fuente: Elaboración Propia

Una vez finalizado la creación del analizador sintáctico se procede a comprobar por medio del analizador semántico que las bases de datos, tablas y campos existan en la misma. A continuación se muestran unos casos de prueba que son verificados por las clases database.php el cual se encarga de constatar que la base de datos y tablas hayan sido creadas y la clase resulset. php comprueba la existencia de columnas. En las siguientes imágenes se ilustra el funcionamiento del análisis semántico.

- Base de datos no existe.

```
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar

?>
```

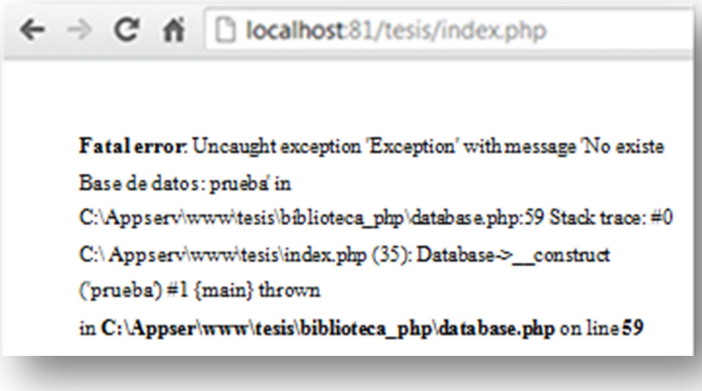

localhost:81/tesis/index.php
Fatal error: Uncaught exception 'Exception' with message 'No existe Base de datos: prueba' in C:\Appserv\www\tesis\biblioteca_php\database.php:59 Stack trace: #0 C:\Appserv\www\tesis\index.php (35): Database->__construct ('prueba') #1 {main} thrown in C:\Appser\www\tesis\biblioteca_php\database.php on line 59

Figura 31 Resultado de BD no existente. Fuente: Elaboración Propia

- Tabla no existe

```php
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar
$db->ejecutar("SELECT campox FROM nueva ",true);

?>
```

Figura 32 Resultado de tabla no existente. Fuente: Elaboración Propia

- Campo no existe

```php
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar
$db->ejecutar("SELECT campox FROM tabla ",true);

?>
```

Figura 33 Resultado de campo no existente. Fuente: Elaboración Propia

Ya finalizada la elaboración del analizador semántico, actualizamos la lista de los requisitos y el siguiente en desarrollarse es el módulo de conexión de las sentencias.

Módulo de Conexión de las Sentencias SQL

Este módulo necesitará de la elaboración de clases y funciones mediante el lenguaje pre-procesado de hipertexto (PHP), las cuales unidas con los archivos generados como son Unetlexer. Php (analizador léxico) y Unetparser.php (analizador sintáctico) podrán verificar la funcionalidad de la biblioteca a través de sentencias Sql.

Se realizó una búsqueda en la red informática mundial (world wide web), en la cual se encontraron códigos desarrollados en lenguaje PHP, que permitían la manipulación de datos a través de archivos, los mismos fueron adaptados para permitir la funcionalidad adecuada según la investigación propuesta. A continuación se enumeran los archivos implementados:

- Util .php
- Stringparser.php
- Sqlquery.php

- Expression.php
- resulSet.php
- config.php
- database.php

En el archivo Util.php, se encuentran las funciones que otras clases requieren para realizar los procesos repetitivos. El segundo archivo Stringparser.php, se encarga de convertir la consulta en un vector manipulable para luego analizar y ejecutar la consulta en los archivos correspondientes, creando objetos del tipo sqlquery para manejar las sentencias en forma de objetos. El archivo sqlquery.php es una clase que tiene los atributos que corresponden a la estructura de una consulta que facilitará su manejo ya que separa en vectores los nombres de columnas, tablas, valores, entre otros. Expression.php se encarga de analizar la sentencia where, y resultSet.php guarda en los vectores los datos leídos o insertados en el archivo de texto. Fueron creados también los archivos config.php que permite la configuración de las rutas donde se localizan los archivos que pertenecen a las base de datos, además se establecen una serie de palabras claves que se usan dentro del código; y database.php se encarga de realizar la interacción con el código del usuario, además las consultas luego de ser evaluados por los analizadores son tomadas por la clase en donde dichos archivos en conjunto interpretan y ejecutan las sentencias Sql en archivos de texto.

En el desarrollo de cualquier sistema o aplicación, para hacer uso de la biblioteca, se debe hacer referencia a la clase database a través de la cláusula include, donde luego se creará una instancia del mismo, enviando como parámetro el nombre de la base de datos a usar; Una vez realizado este proceso, se podrán ejecutar las sentencias requeridas, en el cual el intérprete se encargara de validar las expresiones establecidas por el usuario.

El probador simpletest, es una herramienta que sirve para realizar pruebas unitarias, las cuales permitirán verificar el funcionamiento de las clases o métodos realizados. En la validación de cada uno de ellos, se espera obtener dos posibles respuestas (exitosa o

fallida), de acuerdo si las mismas cumplen o no con la estructura correspondiente.

A continuación se realizan una serie de pruebas:

- La primera prueba se efectuará a la clase principal database.php, y se implementará la clase TestOfDatabase que se ilustra enseguida:

```
<?
require_once "simpletest/autorun.php";
require_once "database.php";
class TestOfDatabase extends UnitTestCase{
   function __construct() {
        parent::__construct("Test database");
    }
   function testEjecutarCreatedatabase(){
     $db= new Database(ROOT_DATABASE);
     $this->assertTrue($db->ejecutar("CREATE DATABASE prueba"));
   }
   function testEjecutarCreatetable(){
     $db= new Database("prueba");
     $this->assertTrue($db->ejecutar("CREATE TABLE tabla (campo1 INTEGER NOT NULL
                                AUTO_INCREMENT,campo2 CHARACTER DEFAULT '1',
                                campo3 INTEGER )"));
}


function testEjecutarInsert(){
   $db= new Database("tesisp");
   $this->assertTrue($db->ejecutar("INSERT INTO join (campo1,campo2) VALUES (15,'q')"));
}
 function testEjecutarUpdate(){
   $db= new Database("tesisp");
   $this->assertTrue($db->ejecutar("UPDATE  join SET campo2='c' WHERE campo1=2"));
 }
 function testEjecutarDelete(){
   $db= new Database("tesisp");
   $this->assertTrue(is_numeric($db->ejecutar("DELETE FROM  join  WHERE campo1=4")));
 }
 function testEjecutarSelect(){
   $db= new Database("tesisp");
   $this->assertTrue($db->ejecutar("SELECT * FROM join") instanceof ResultSet);
 }

 function testEjecutarDroptable(){
   $db= new Database("prueba");
   $this->assertTrue($db->ejecutar("DROP TABLE tabla"));
 }
```

```
   function testEjecutarDropdatabase(){
     $db= new Database(ROOT_DATABASE);
     $this->assertTrue($db->ejecutar("DROP DATABASE prueba"));
   }
}
?>
```

El resultado obtenido durante las pruebas realizadas a la clase Database se dieron de manera exitosa, donde se realizó una sola ejecución para las diferentes funciones requeridas en el manejo de datos, a continuación se muestra en la figura 34 lo que se obtuvo con la aplicación “simpletest”:

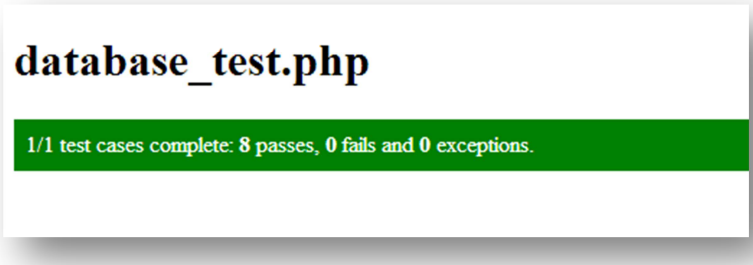


Figura 34 Resultado de la clase database. Fuente: Elaboración Propia

- La segunda prueba se realizará a la clase Interprete, la cual está constituida de la siguiente manera:

```
<?
require_once "simpletest/autorun.php";
require_once "util.php";
require_once "StringParser.php";
class TestOfStringParser extends UnitTestCase{
   function __construct() {
        parent::__construct("Test stringparser");
    }
   function teststringparser(){
     $sp= new StringParser ("SELECT * FROM tabla");
     //SELECT 1
     // *     2
     //FROM   3
     //tabla  4
     $this->assertTrue(count($sp->actual)==4);
   }
   function testparsequeryselect(){
```

```
    $sp= new StringParser ("SELECT * FROM tabla");
    $retorno=$sp->parsequery();
    $this->assertTrue(strcmp($retorno->type,"SELECT")==0);
  }
  function testparsequeryinsert(){
    $sp= new StringParser ("INSERT INTO tabla(campo1,campo2) VALUES (1,'2')");
    $retorno=$sp->parsequery();
    $this->assertTrue(strcmp($retorno->type,"INSERT")==0);
  }
  function testparsequeryinsertcampos(){
    $sp= new StringParser ("INSERT INTO tabla(campo1,campo2) VALUES (1,'2')");
    $retorno=$sp->parsequery();
    $this->assertTrue(in_array("campo1",$retorno->colNames));
    $this->assertTrue(in_array("campo2",$retorno->colNames));
  }
  function testparsequeryupdate(){
    $sp= new StringParser ("UPDATE tabla SET campo1=2 WHERE campo2=1");
    $retorno=$sp->parsequery();
    $this->assertTrue(strcmp($retorno->type,"UPDATE")==0);
  }
   function testparsequerydelete(){
    $sp= new StringParser ("DELETE FROM tabla WHERE campo2=1");
    $retorno=$sp->parsequery();
    $this->assertTrue(strcmp($retorno->type,"DELETE")==0);
  }
}

?>
```

Posterior a dicha prueba, se obtuvo un resultado satisfactorio, la cual se ilustra en la figura 35:

Figura 35. Resultado de la clase interprete. Fuente: Elaboración Propia

- Para llevar a cabo la tercera prueba "resultset" es necesario hacer uso del archivo que se muestra en la figura 36:

```
campo1#campo2#campo3#campo4
auto_increment#character#character#integer
#1##
1#1#a#4
2#1#b#4
3#1#c#4
4#1#d#3
5#1#e#4
6#1#g#4
7#1#x#10
8#1#p#54
```

Figura 36 Archivo de ejemplo. Fuente: Elaboración Propia

```php
<?
require_once "simpletest/autorun.php";
require_once "database.php";
class TestOfResultSet extends UnitTestCase{
    function __construct() {
        parent::__construct("Test resultset");
    }
  function testresulset(){
     $db =new Database("tesisp");
     $rs=$db->readTable("prueba");//Devuelve objeto ResultSet
     $row=$rs->rows[0];
     $this->assertTrue($row instanceof Row);
     $this->assertTrue( count($rs->colNames)==4);//Existen 4 campos en el archivo
     $this->assertTrue( $rs->getRowCount()==8);//Existen 8 registros en el archivo
     $this->assertTrue( count($row->fields)==4);//Existen 4 campos en el row
  }
}
?>
```

En la figura 37 se ilustra el buen resultado que se obtuvo de la ejecución de la aplicación simpletest:

Figura 37 Resultado de la clase resultset. Fuente: Elaboración Propia

Una vez realizada una breve explicación de cada clase y aplicada la prueba con la herramienta simpletest, se muestran a continuación unos ejemplos que permitirán hacer uso

de la biblioteca php, ejecutando sentencias SQL, para la inserción, modificación y eliminación de datos en los archivos de texto que emularan las tablas de una Base de Datos para la administración de archivos.

Para crear o eliminar la base de datos es necesario usar la instancia de ROOT_DATABASE, como se puede observar en el siguiente código:

```
<?php
      include_once 'database.php';

$db=new Database(ROOT_DATABASE)
      /*Las sentencias a ejecutar retornan true o false con excepción
      del SELECT que retorna un ResultSet,
DELETE y UPDATE que retornan numero de filas afectadas*/
if($db->ejecutar("CREATE DATABASE prueba",true))
 echo "<br/>BD creada con exito";

?>
```

El resultado es la creación de la base de datos, el cual es una carpeta con el nombre que en este caso sería “prueba”, se verá en la figura 38:

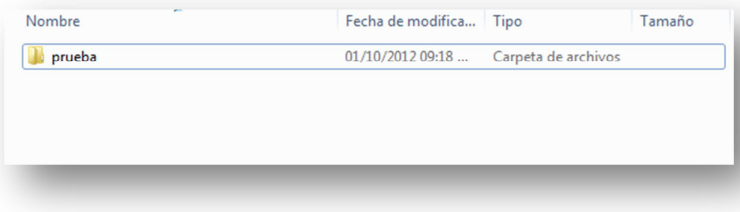


Figura 38 Resultado de creación base de datos. Fuente: Elaboración Propia

Luego de haber creado la base de datos con el nombre de “prueba”, el siguiente paso es la creación de la tabla por medio de la sentencia CREATE TABLE, la tabla estará representada por un archivo de texto, la cual será encontrada dentro de la carpeta prueba, a continuación se muestra el código:

```
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar
if($db->ejecutar("CREATE TABLE tabla (campo1 INTEGER NOT NULL AUTO_INCREMENT,campo2
                  CHARACTER DEFAULT '1',campo3 INTEGER)",true))
            echo "<br/>Tabla creada con éxito";
```

```
?>
```

Como se mencionó anteriormente, la tabla creada en un archivo de texto.

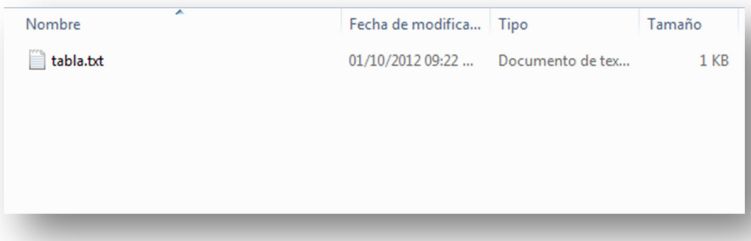


Figura 39 Resultado de creación de la tabla. Fuente: Elaboración Propia

Después de haber creado la tabla, dentro de la misma insertamos los registros por medio de la sentencia INSERT INTO, ver código a continuación:

```
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar

for($i=0;$i<3;$i++){
        if($db->ejecutar("INSERT INTO tabla (campo3) VALUES ($i)",true)){
        $id= $db->getLastInsertId();//Retorna Ultimo Id insertado
                echo "<br/>Insertado registro con id: $id";
        }
}
?>
```

Se obtiene como resultado la inserción del nombre de los campos, los tipos de datos y el valor que se asignó por medio del ciclo realizado.

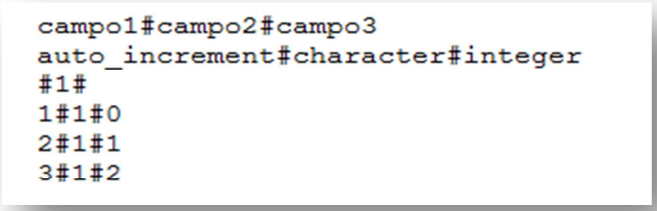


Figura 40 Resultado de inserción en la tabla. Fuente: Elaboración Propia

Luego se podrá seleccionar los datos por medio de la sentencia sql SELECT.

```
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar

$db->ejecutar("SELECT * FROM tabla",true);
while($db->next_record()){
                echo "<br/> FILA ->".$db->f(0)." - ".$db->f(1)." - ".$db->f(2);
}
?>
```

De esta manera los registros seleccionados durante la ejecución del select, son todos los datos almacenados, como se ilustra a continuación:

FILA ->1 – 1 - 0
FILA ->2 – 1 - 1
FILA ->3 - 1 - 2

Se podrá realizar la eliminación de un registro que se encuentre dentro de la tabla por medio de la sentencia DELETE, el siguiente código muestra cómo realizar la ejecución de la sentencia:

```
<?php
```

```php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar

if($ret=$db->ejecutar("DELETE FROM tabla WHERE campo1=1",true)!=0)
                echo "<br/>$ret Filas eliminadas";
?>
```

Observemos que se ha eliminado el registro correspondiente a la fila 1, que una variable incremental, como se asignó en la sentencia delete.

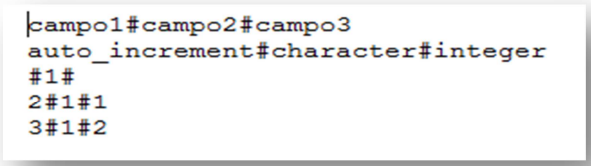


Figura 41 Resultado de eliminación en la tabla. Fuente: Elaboración Propia

Otro ejemplo de la sentencia Select, luego de eliminar el registro de campo1, a continuación ver código con su resultado:

```php
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar
$db->ejecutar("SELECT campo1,campo2 FROM tabla",true);
while($db->next_record()){
                echo "<br/> FILA ->".$db->f(0)." - ".$db->f("campo2");
}
?>
```

FILA ->2 - 1
FILA ->3 - 1

También se podrá hacer uso de la sentencia Update, donde se pueden modificar registros, como se muestra en esta parte de código, cambiando el valor del campo2 a cero con la condición de que el campo1 sea igual a 2.

```php
<?php
```

```php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar
if($ret=$db->ejecutar("UPDATE tabla SET campo2='0' WHERE campo1=2",true)!=0)
        echo "<br/>$ret Filas actualizadas";
?>
```

El resultado de la modificación del campo2 igual a cero se muestra en la figura 42:

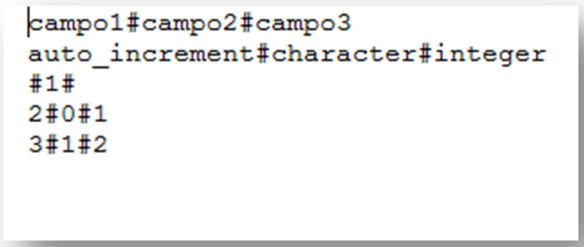


Figura 42 Resultado de la modificación en la tabla. Fuente: Elaboración Propia

Se ilustra otro ejemplo de la sentencia select con su resultado.

```php
<?php
include_once 'database.php';

$db=new Database("prueba");//Nombre de la BD a usar
$db->ejecutar("SELECT campo1,campo2 FROM tabla  WHERE campo1=2",true);
while($db->next_record()){
        echo "<br/> FILA ->".$db->f(0)." - ".$db->f("campo2");
}  ?>
```

FILA ->2 - 0

Como se puede observar, con el uso de los archivos realizados en php junto con la elaboración de las sentencias sencillas de sql como creación, inserción, modificación y eliminación, se constató el buen funcionamiento de la biblioteca php, mediante la gestión de archivos de texto.

Ya finalizada la construcción de las distintas clases con sus funciones, actualizamos la lista teniendo como requisito el módulo de acceso a usuario, en el cual se realizará una pequeña aplicación.

Módulo de Acceso a Usuario

La finalidad de elaborar la biblioteca php es la de mejorar los tiempos de respuestas, además de ofrecer a los usuarios facilidad, sencillez al momento de uso, del mismo modo la biblioteca tiene como propósito el de imitar un motor de almacenamiento, permitiendo realizar la búsqueda de registros de manera eficaz. El programador al momento de hacer uso de la biblioteca php, debe guardar el archivo de la biblioteca dentro de la carpeta donde desarrolle su proyecto de manera que pueda extender la biblioteca y no presente inconvenientes al momento de ejecutar una sentencia. Para tener una información completa sobre el uso de la biblioteca debe leer el manual de usuario que se encontrará en los anexos.

Validación de la Biblioteca.

Para verificar o certificar el funcionamiento de la biblioteca elaborada en el lenguaje pre-procesado de hipertexto (PHP), se realizará una pequeña aplicación donde se comprobará la ejecución de las distintas sentencias sql mediante una interfaz Web donde se pueda crear base de datos, posterior a dicho proceso la creación de tablas, y una interfaz donde al seleccionar la misma permitirá realizar la inserción, modificación y eliminación de datos.

A continuación se muestran los diferentes pasos a seguir por el programador para comprobar la funcionalidad de la biblioteca que fue incluida en una pequeña aplicación elaborada en php:

- En la ventana de inicio de sesión se pide el login y password con el fin de autenticar el usuario.

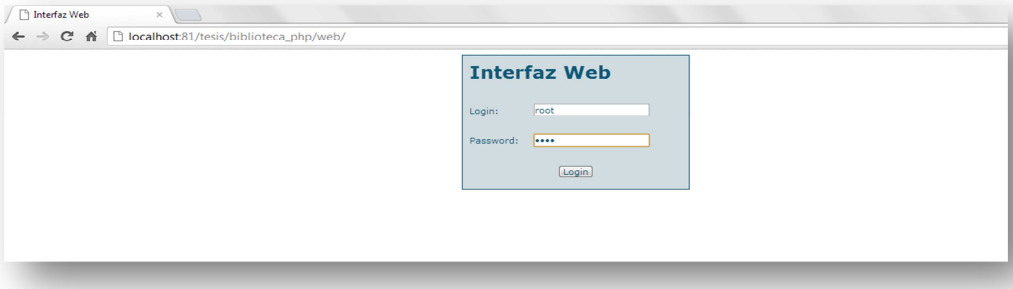


Figura 43 Inicio de sesión. Fuente: Elaboración Propia

- Luego del usuario haber iniciado sesión, aparece la interfaz (figura 44), donde al lado izquierdo se encuentran las bases de datos creadas y en el centro de la pantalla se puede observar donde se introduce el nombre con el cual el usuario quiere crear su base de datos.

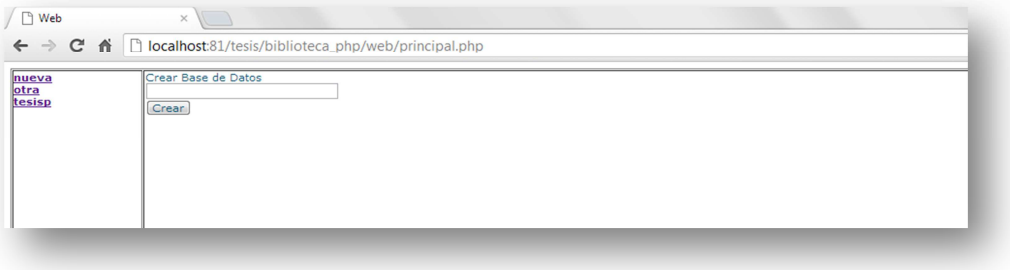


Figura 44 Crear base de datos. Fuente: Elaboración Propia

- Una vez creada la base de datos, se crea la tabla donde se escribe el nombre y los campos como se muestra en la figura 45:

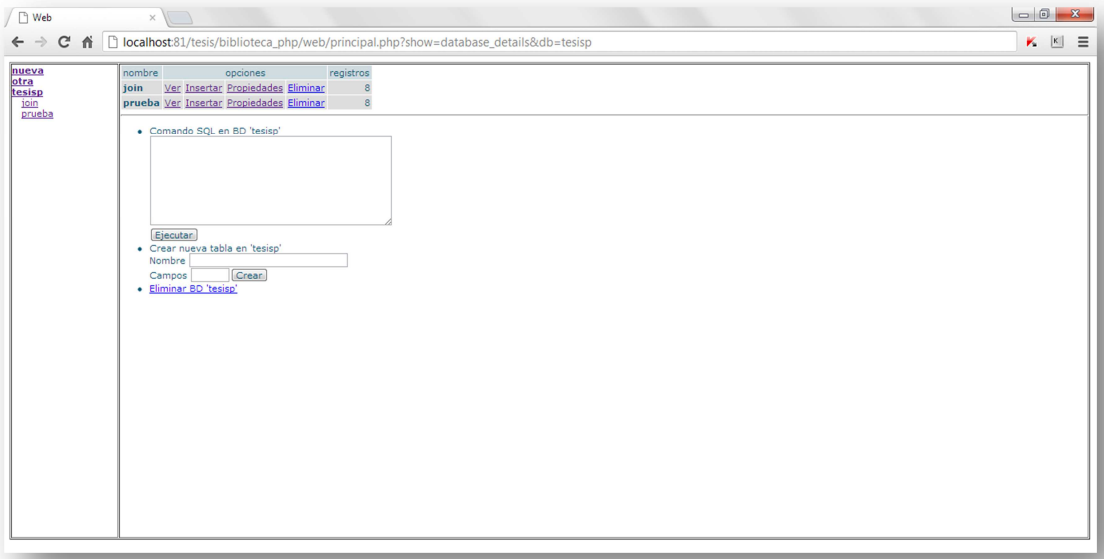


Figura 45 Crear tabla, con nombre y campos. Fuente: Elaboración Propia

- Después de ingresar el nombre y los campos de la base de datos, se procede a identificar cada campo colocando el nombre y el tipo de dato a utilizar.

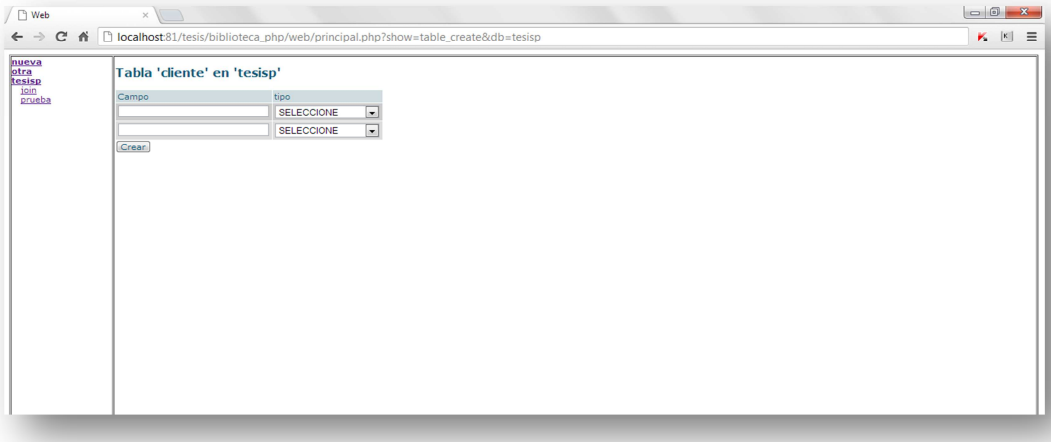


Figura 46 Ingresar nombre y tipo de dato. Fuente: Elaboración Propia

- En la siguiente imagen se puede observar los distintos registros encontrados en la tabla.

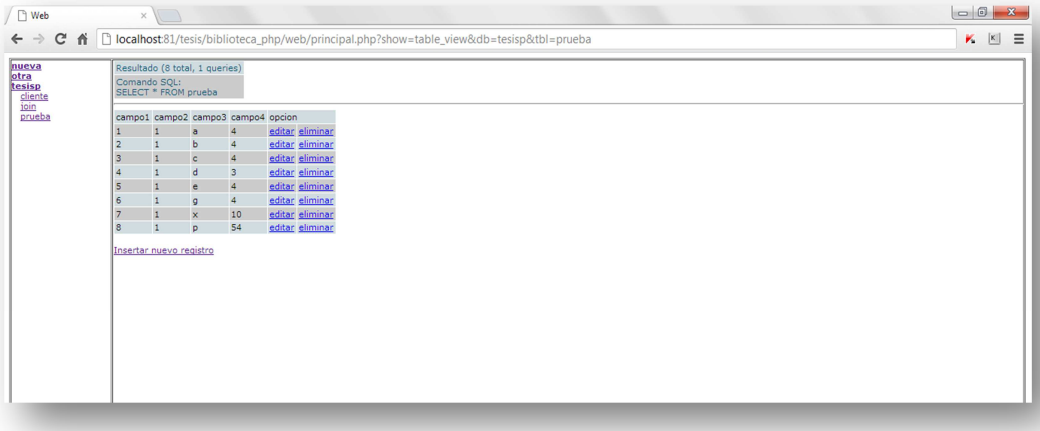


Figura 47 Registros de la tabla. Fuente: Elaboración Propia

- En la figura 48, se puede observar la interfaz para introducir los valores a insertar.

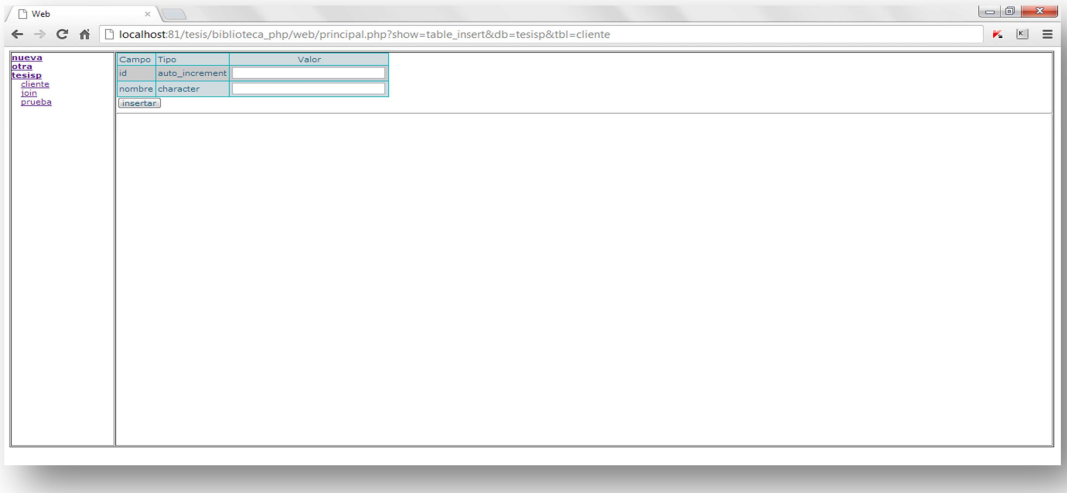


Figura 48 Insertar valor en la tabla. Fuente: Elaboración Propia

- La opción propiedades muestra los campos y tipos de datos.

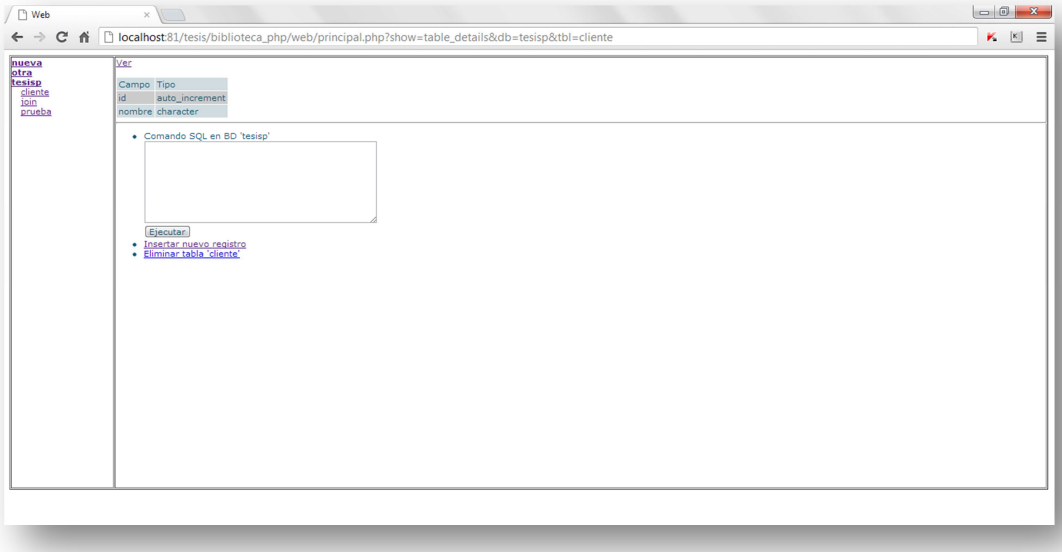


Figura 49 Opción propiedades. Fuente: Elaboración Propia

- En la figura 50, se muestra el mensaje de confirmación al momento de eliminar una tabla, así mismo seria para la eliminación de base de datos y de registros.

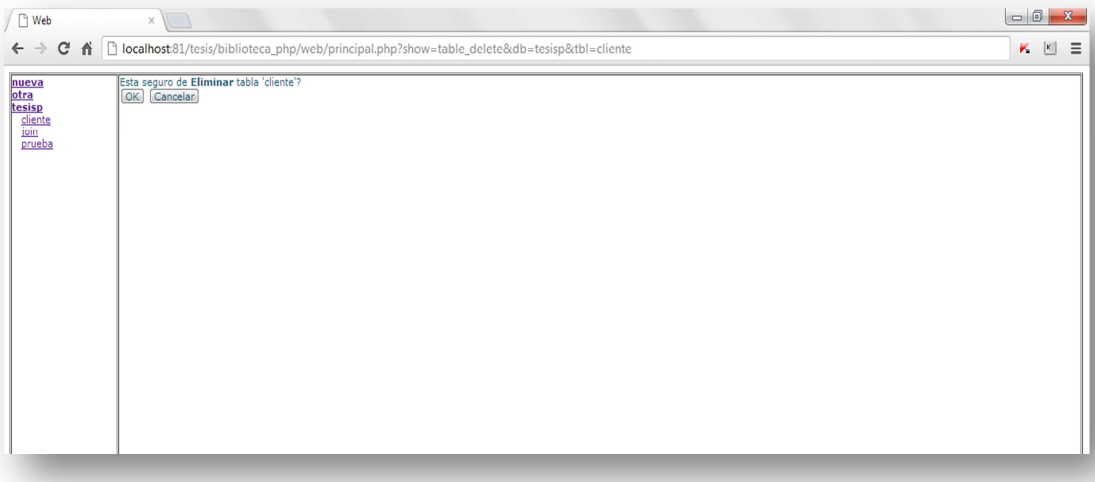


Figura 50 Eliminación de la tabla. Fuente: Elaboración Propia